LambdaCube 3D

purely functional API for GPU graphics
http://lambdacube3d.com

Csaba Hruska

Better graphics programming

Goals

- Less errors
- More code reuse
- Keep efficiency

Ideas

- Declarative description

  Dataflow based graphics pipeline model
- Compile time validation

  Use clever type system to check API constraints

Automate the engine coder’s work as much as possible.

Imperative GPU graphics programming

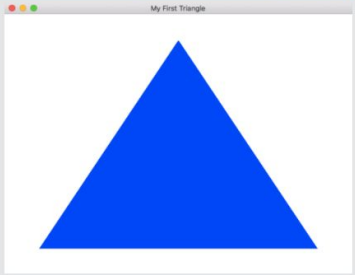
```haskell
initResources :: IO Program
initResources = do
    -- compile vertex shader
    vs <- U.loadShader GL.VertexShader "triangle.v.glsl"
    fs <- U.loadShader GL.FragmentShader "triangle.f.glsl"
    p <- U.linkShaderProgram [vs, fs]
    GL.blend $= GL.Enabled
    GL.blendFunc $= (GL.SrcAlpha, GL.OneMinusSrcAlpha)
    Program p <$> GL.get (GL.attribLocation p "coord2d")

draw :: Program -> GLFW.Window -> IO ()
draw (Program program attrib) win = do
    GL.clearColor $= GL.Color4 1 1 1 1
    GL.clear [GL.ColorBuffer]
    (width, height) <- GLFW.getFramebufferSize win
    GL.viewport $= (GL.Position 0 0, GL.Size (fromIntegral width) (fromIntegral height))

    GL.currentProgram $= Just program
    GL.vertexAttribArray attrib $= GL.Enabled
    V.unsafeWith vertices $ \ptr ->
        GL.vertexAttribPointer attrib $=
          (GL.ToFloat, GL.VertexArrayDescriptor 2 GL.Float 0 ptr)
    GL.drawArrays GL.Triangles 0 3 -- 3 is the number of vertices
    GL.vertexAttribArray attrib $= GL.Disabled
```

OpenGL in general

Resource allocation
1. upload data to GPU
2. upload programs (shaders) to GPU

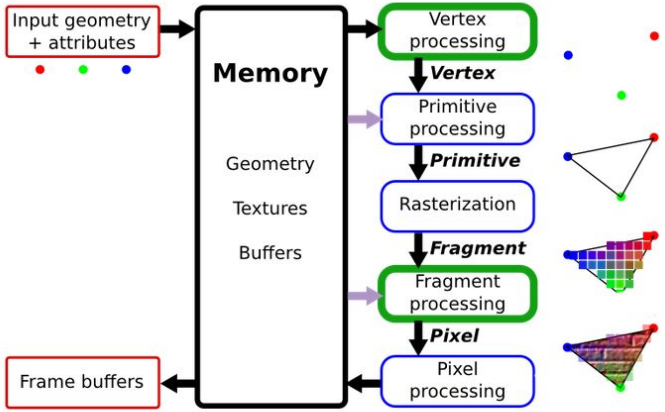
Setup draw state
3. setup rendering features
   (i.e. *blending*, *clipping*)
4. attach input/output buffers

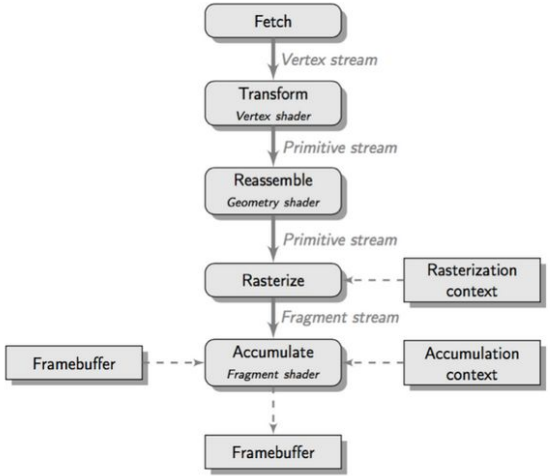
Draw call
5. execute drawing commands
6. goto 3 (optional)
   i.e. *multipass rendering*



Dataflow Model = Functional Programming

Treat the GPU configuration state as a parameter for each draw command

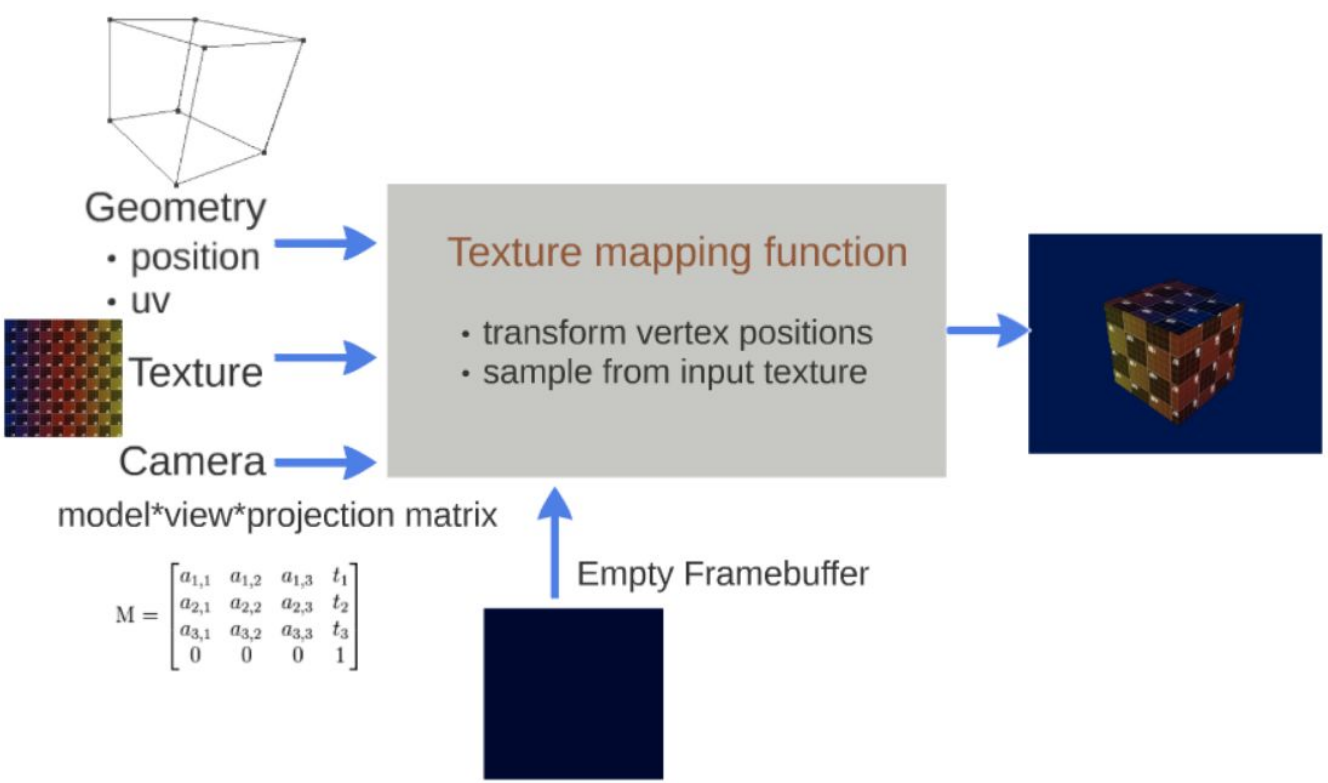


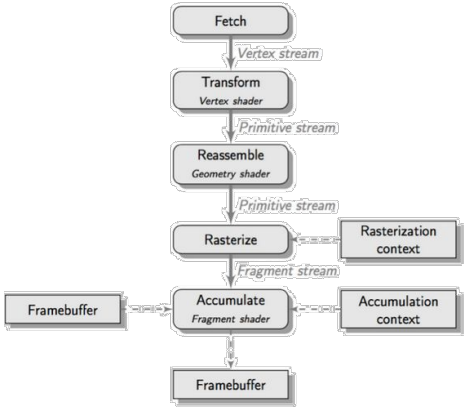


Collect the relevant OpenGL state parts that has effect on draw operations e.g.

- used *vertex* and *fragment* shader
- configuration for *rasterization* (Rasterization context)
- configuration for *pixel processing* (Accumulation context)

Example: Texture mapping pipeline





LambdaCube 3D

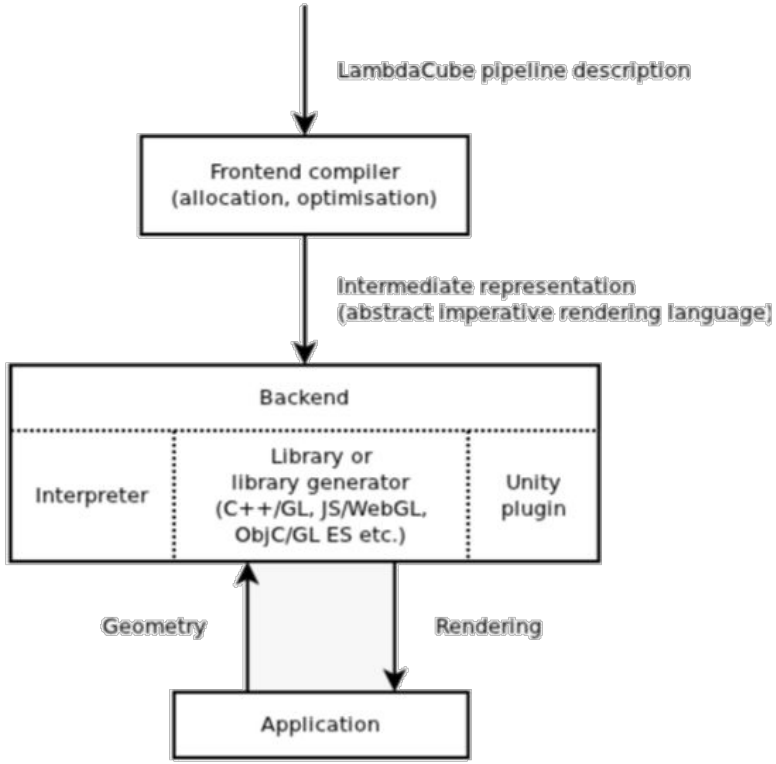
LambdaCube 3D is Haskell-like purely functional domain specific language for programming the GPU.

http://lambdacube3d.com/

frontend: lambdacube-compiler

backends:

- lambdacube-gl (*Haskell, Desktop*)
- purescript-lambdacube-webgl (*PureScript, Web*)
- android-gles20 (*Java, Android, experimental*)
- ios-gles20 (*C++, iOS, experimental*)



LambdaCube 3D

http://lambdacube3d.com/

Purely functional GPU graphics API

- Dataflow based declarative description
- Compile-time validation of GPU API constraints via types
- Better code reuse via function composition